SUBSTITUTE SPECIFICATION

CLEAN VERSION
FOR ENTRY

Attorney Docket No. SUN1P253/P4194

APPLICATION FOR U.S. PATENT

MULTI-LINGUAL TAG EXTENSION MECHANISM

INVENTORS: Eduardo Pelegri-Llopart
413 Gilbert Avenue
Menlo Park, Ca 94025

A citizen of Spain

Laurence P. G. Cable

A citizen of the United Kingdom

ASSIGNEE: SUN MICROSYSTEMS, INC.
901 SAN ANTONIO ROAD
PALO ALTO, CA 94303

A DELAWARE CORPORATION

ENTITY: LARGE

BEYER & WEAVER, LLP
P.O. Box 61059
Palo Alto, CA 94306
Telephone (650) 493-2100

SDB/DBJ

MULTI-LINGUAL TAG EXTENSION MECHANISM

BACKGROUND OF THE INVENTION

[0001] This application claims priority from U.S. Provisional Application Number 60/141,071, filed June 25, 1999, Attorney Docket No. SUN1P249P/P4205PSP, entitled "JAVA SERVER PAGES SPECIFICATION"; and claims priority from U.S. Provisional Application Number 60/149,508, filed August 17, 1999, Attorney Docket No. SUN1P260P, entitled "JAVASERVER PAGES SPECIFICATION"; and is related to U.S. Patent Application Number 09/471,072 entitled "MECHANISM FOR AUTOMATIC SYNCHRONIZATION OF SCRIPTING VARIABLES" attorney docket number SUN1P254/P4195, filed December 21, 1999, the applications are commonly assigned to the assignee of the present invention, and the disclosures of which are herein incorporated by reference.

1. Field of the Invention

[0002] The present invention relates generally to the field of computer software, and more particularly to a multi-lingual tag extension mechanism suitable for use with JAVASERVER™ Pages.

2. Description of the Related Art

[0003] JAVA™ programming language is expressly designed for use in the distributed environment of the Internet. It was designed to have the "look and feel" of the C++ language, but it is simpler to use than C++ and enforces an object-oriented programming model. JAVA™ programming language can be used to create complete applications that may run on a single computer or be distributed among servers and clients in a network. It can also be used to build a small application module or applet for use as part of a Webpage. Applets make it possible for a Web page user to interact with the page. JAVASERVER™ Pages is the JAVA™ platform technology for building applications containing dynamic Web content

such as HTML, DHTML, XHTML, and XML. A JAVASERVER™ Page (JSP) is a text-based document that describes how to process a request to create a response. The description inter-mixes template data with some dynamic actions, taking advantage of the capabilities of the JAVA™ platform. The template data is commonly fragments of a structured document (HTML, DHTML, XHTML or XML), and the dynamic actions can be described with scripting elements and/or server-side actiontags.

[0004] A simple example of a JSP page is shown in Figure 2. The example shows the response page, which is intended to be a short list of the day of the month and year, at the moment the request is received by the server. The page itself contains some fixed template text, and JSP elements that are shown underlined in the figure. The underlined actions are executed on the server side. When a client makes a request, such as an HTTP request, a request object requests a response from the JAVASERVER™ container. The first element creates a JAVA BEAN™ named clock, of type calendar.jspCalendar. The next two elements use the Bean to display some of its properties (i.e. month and year). The output is sent to a response object which sends a response back to the client.

[0005] A JSP page is executed by a JSP container, which is installed on a Web server, or on a Web enabled application server. The JSP container delivers requests from a client to a JSP page and responses from the JSP page to the client. JSP pages may be implemented using a JSP translation or compilation phase that is performed only once, followed by a request processing phase that is performed once per request. The translation phase creates a JSP page implementation class that implements a servlet interface.

[0006] Typically, a JSP page contains declarations, fixed template data, action instances that may be nested, and scripting elements. When a request is delivered to a JSP page, all these components are used to create a response object that is then returned to the client. As with standard Web pages, JSP pages may contain "tags." In order to facilitate the

and placement of graphic elements, and also providing links to interactive programs and scripts.

[0007] In standard implementations, a JSP page is translated into JAVA™ code that runs on a server. The transformation from the JSP page into JAVA™ code is done once, when the page is first accessed. The JAVA™ code running on the server is activated when a request is received. In order to create a response object, certain data is passed verbatim to the response, and some actions are executed on the server side. Some of these actions are explicitly spelled out in the JSP page, and other actions are described by the semantics of the text. Thus, there are three types of code: verbatim code, scripting code from the JSP page, and code which has to be defined by the tag library. As used herein, "tags" in the JSP context will be referred to as "actions."

[0008] Ideally, a tag mechanism employed in a JSP page system would allow for actions to be defined in standard libraries. This would allow third parties to provide new actions as part of their JSP authoring tools. For example, as shown in Figure 3, a JSP page author may use a JSP specification-compliant authoring tool to create a Web page. The vendor of the authoring tool can provide new actions via a JSP tag library, such as a tag library that supports chat room functionality. The page can then be deployed into any JSP-compliant container, such as a Web browser. The Web browser uses the same tag library information in order to run the Web page, including the desired chat room functions.

[0009] In other words, if a standard JSP tag mechanism is properly defined, vendors of tag libraries can use the standard specification to create tag libraries that are compliant with the JSP environment. Also, vendors of authoring tools can create authoring tools (and scripting languages) compliant with the specification, and vendors of Web browsers can create JSP compliant Web browsers. A Web page author can then choose the best tag library and the best authoring tool available for creating the desired Web pages. Two different Web

browsers may support scripting in a completely different manner, but the same tag libraries must be supported by both in order to run the Web page.

[0010] Therefore, in order to provide the desired functionality, the JSP specification needs to define the semantics of an extensible tag mechanism that can support the following constraints:

- the tag mechanism can be implemented with minimal assumptions about the page implementation (i.e. only "callability" from the page into the extension should be assumed);
- minimal constraints on the page translation process are imposed;
- supports tag body evaluation;
- portable - a tag described in a tag library must be usable in any JSP container;
- simple - a mechanism that is accessible and easy to use, both by users of the tags (JSP authors, creators of JSP authoring tools) and by authors of tags;
- expressive - enable a wide range of actions; in particular:
  - nested actions
  - scripting elements inside the tag body
  - refer to scripting variables, and create and update scripting variables; and
- usable from different scripting languages, not just JAVA™.

The more powerful the tag library mechanism, however, the more difficult it is to satisfy these constraints.

[0011] Currently available commercial products do not provide the desired functionality. For example, COLD FUSION's™ CFX mechanism does not provide for body evaluations. Compilation-based mechanisms, such as LIVESOFTWARE's™ JRUN™ mechanism, make significant assumptions about the page implementation details. Also, pure translation-based methods, like XSLT need to make assumptions about the scripting language.

[0012] One prior art approach is to define the semantics of each tag through a transformation into a scripting language. The problem with this approach, however, is that the tag library author does not know the scripting language that will be used, so it is very difficult to define the transformation. A second approach relies on the concept of a "closure." In an evaluation system, a closure is a data structure that holds an expression and an environment of variable bindings in which that expression is to be evaluated. The variables may be local or global. Closures are used to represent unevaluated expressions when implementing functional programming languages with lazy evaluation.

[0013] Figure 4 illustrates how a tag can be translated into a closure. The idea of a closure is to encapsulate a tag "body" into a software object called a closure. A closure is a procedure with some context, such that the object can evaluate itself. The closure is instantiated and passed to an implementation of the tag library. The tag library can then execute the action and determine how many times to invoke the body if, for example, the body contains a loop.

[0014] During the translation process, a tag "foo" in a JSP page (Figure 4) is transformed into a closure. An Implementation Class (type servlet) is created, with the code associated with the "Body" of the tag corresponding to the closure (i.e. the underlined code). When a request is received, an instance of the closure is passed to the tag library. The tag library then executes the action, using a single "doIt ( )" method. The main advantage of using a closure to perform the tag action is that no special knowledge is needed in order to execute the closure. Specifically, no knowledge is required of how variables are accessed, since the closure contains the necessary context. A closure is difficult to create, however, since it is difficult to provide the requisite context. In fact, the JAVA™ programming language does not support the use of closures.

[0015] In view of the foregoing, there is a need for an extensible tag mechanism that can provide the desired functionality in JAVA™, regardless of the scripting language used to develop the JSP page (i.e. multi-lingual), and which does not use a standard closure.

SUMMARY OF THE INVENTION

[0016] In general, the present invention is a multi-lingual tag extension mechanism suitable for use with the JAVA SERVER™ Pages platform. Rather than creating a closure abstraction, the present invention "in-lines" a body evaluation for the tags (i.e. actions). A doStart ( ) method processes a start tag and determines if a body needs to be evaluated. If so, a body evaluation buffer is passed to a doBody ( ) method for body evaluation. Once the body evaluation is completed, a doEnd ( ) method completes the processing by synchronizing the variables. The scripting details of the present invention match any nesting of the tag, so that the structure corresponding to the original scripting is preserved. Furthermore, the present invention is not dependent on the specifics of the scripting language in the JSP page used to form the response.

[0017] The present invention thus enables the addition of new or custom actions, allowing the JSP "language" to be easily extended in a portable fashion. Tag libraries can be used by JSP authoring tools and can be distributed along with JSP pages to any JSP container, such as Web and application servers. The tag extension mechanism of the present invention can be used from JSP pages written using any valid scripting language, although the mechanism itself only assumes a JAVA™ RunTime environment.

BRIEF DESCRIPTION OF THE DRAWINGS

[0018] The present invention will be readily understood by the following detailed description in conjunction with the accompanying drawings, wherein like reference numerals designate like structural elements, and in which:

Figure 1 is a block diagram of a computer system suitable for implementing the present invention;

Figure 2 illustrates a JSP page;

Figure 3 is a diagram illustrating the relationship between a tag library, a JSP page, and a Web browser;

Figure 4 illustrates an implementation of a closure;

Figure 5 illustrates an implementation of the present invention; and

Figure 6 illustrates the code of Figure 4 translated according to the present invention.

DETAILED DESCRIPTION OF THE INVENTION

[0019] The following description is provided to enable any person skilled in the art to make and use the invention and sets forth the best modes contemplated by the inventor for carrying out the invention. Various modifications, however, will remain readily apparent to those skilled in the art, since the basic principles of the present invention have been defined herein specifically to provide a multi-lingual tag extension mechanism.

[0020] The present invention employs various computer-implemented operations involving data stored in computer systems. These operations include, but are not limited to, those requiring physical manipulation of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. The operations described herein that form part of the invention are useful machine operations. The manipulations performed are often referred to in terms, such as, producing, identifying, running, determining, comparing, executing, downloading, or detecting. It is sometimes convenient, principally for reasons of common usage, to refer to these electrical or magnetic signals as bits, values, elements, variables, characters, data, or the like. It should remembered, however, that all of these and

similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities.

[0021] The present invention also relates to a device, system or apparatus for performing the aforementioned operations. The system may be specially constructed for the required purposes, or it may be a general purpose computer selectively activated or configured by a computer program stored in the computer. The processes presented above are not inherently related to any particular computer or other computing apparatus. In particular, various general-purpose computers may be used with programs written in accordance with the teachings herein, or, alternatively, it may be more convenient to construct a more specialized computer system to perform the required operations.

[0022] FIG. 1 is a block diagram of a general purpose computer system 100 suitable for carrying out the processing in accordance with one embodiment of the present invention. Figure 1 illustrates one embodiment of a general purpose computer system. Other computer system architectures and configurations can be used for carrying out the processing of the present invention. Computer system 100, made up of various subsystems described below, includes at least one microprocessor subsystem (also referred to as a central processing unit, or CPU) 102. That is, CPU 102 can be implemented by a single-chip processor or by multiple processors. It should be noted that in re-configurable computing systems, CPU 102 can be distributed amongst a group of programmable logic devices. In such a system, the programmable logic devices can be reconfigured as needed to control the operation of computer system 100. In this way, the manipulation of input data is distributed amongst the group of programmable logic devices. CPU 102 is a general purpose digital processor which controls the operation of the computer system 100. Using instructions retrieved from memory, the CPU 102 controls the reception and manipulation of input data, and the output and display of data on output devices.

[0023] CPU 102 is coupled bi-directionally with a first primary storage 104, typically a random access memory (RAM), and uni-directionally with a second primary storage area 106, typically a read-only memory (ROM), via a memory bus 108. As is well known in the art, primary storage 104 can be used as a general storage area and as scratch-pad memory, and can also be used to store input data and processed data. It can also store programming instructions and data, in the form of data objects, in addition to other data and instructions for processes operating on CPU 102, and is used typically used for fast transfer of data and instructions in a bi-directional manner over the memory bus 108. Also as well known in the art, primary storage 106 typically includes basic operating instructions, program code, data and objects used by the CPU 102 to perform its functions. Primary storage devices 104 and 106 may include any suitable computer-readable storage media, described below, depending on whether, for example, data access needs to be bi-directional or uni-directional. CPU 102 can also directly and very rapidly retrieve and store frequently needed data in a cache memory 110.

[0024] A removable mass storage device 112 provides additional data storage capacity for the computer system 100, and is coupled either bi-directionally or uni-directionally to CPU 102 via a peripheral bus 114. For example, a specific removable mass storage device commonly known as a CD-ROM typically passes data uni-directionally to the CPU 102, whereas a floppy disk can pass data bi-directionally to the CPU 102. Storage 112 may also include computer-readable media such as magnetic tape, flash memory, signals embodied on a carrier wave, PC-CARDS, portable mass storage devices, holographic storage devices, and other storage devices. A fixed mass storage 116 also provides additional data storage capacity and is coupled bi-directionally to CPU 102 via peripheral bus 114. The most common example of mass storage 116 is a hard disk drive. Generally, access to these media is slower than access to primary storages 104 and 106.

[0025] Mass storage 112 and 116 generally store additional programming instructions, data, and the like that typically are not in active use by the CPU 102. It will be appreciated that the information retained within mass storage 112 and 116 may be incorporated, if needed, in standard fashion as part of primary storage 104 (e.g. RAM) as virtual memory.

[0026] In addition to providing CPU 102 access to storage subsystems, the peripheral bus 114 is used to provide access other subsystems and devices as well. In the described embodiment, these include a display monitor 118 and adapter 120, a printer device 122, a network interface 124, an auxiliary input/output device interface 126, a sound card 128 and speakers 130, and other subsystems as needed.

[0027] The network interface 124 allows CPU 102 to be coupled to another computer, computer network, or telecommunications network using a network connection as shown. Through the network interface 124, it is contemplated that the CPU 102 might receive information, *e.g.*, data objects or program instructions, from another network, or might output information to another network in the course of performing the above-described method steps. Information, often represented as a sequence of instructions to be executed on a CPU, may be received from and outputted to another network, for example, in the form of a computer data signal embodied in a carrier wave. An interface card or similar device and appropriate software implemented by CPU 102 can be used to connect the computer system 100 to an external network and transfer data according to standard protocols. That is, method embodiments of the present invention may execute solely upon CPU 102, or may be performed across a network such as the Internet, intranet networks, or local area networks, in conjunction with a remote CPU that shares a portion of the processing. Additional mass storage devices (not shown) may also be connected to CPU 102 through network interface 124.

[0028] Auxiliary I/O device interface 126 represents general and customized interfaces that allow the CPU 102 to send and, more typically, receive data from other devices such as microphones, touch-sensitive displays, transducer card readers, tape readers, voice or handwriting recognizers, biometrics readers, cameras, portable mass storage devices, and other computers.

[0029] Also coupled to the CPU 102 is a keyboard controller 132 via a local bus 134 for receiving input from a keyboard 136 or a pointer device 138, and sending decoded symbols from the keyboard 136 or pointer device 138 to the CPU 102. The pointer device may be a mouse, stylus, track ball, or tablet, and is useful for interacting with a graphical user interface.

[0030] In addition, embodiments of the present invention further relate to computer storage products with a computer readable medium that contain program code for performing various computer-implemented operations. The computer-readable medium is any data storage device that can store data which can thereafter be read by a computer system. The media and program code may be those specially designed and constructed for the purposes of the present invention, or they may be of the kind well known to those of ordinary skill in the computer software arts. Examples of computer-readable media include, but are not limited to, all the media mentioned above: magnetic media such as hard disks, floppy disks, and magnetic tape; optical media such as CD-ROM disks; magneto-optical media such as floptical disks; and specially configured hardware devices such as application-specific integrated circuits (ASICs), programmable logic devices (PLDs), and ROM and RAM devices. The computer-readable medium can also be distributed as a data signal embodied in a carrier wave over a network of coupled computer systems so that the computer-readable code is stored and executed in a distributed fashion. Examples of program code include both machine code, as produced, for example, by a compiler, or files containing higher level code that may be executed using an interpreter.

[0031] It will be appreciated by those skilled in the art that the above described hardware and software elements are of standard design and construction. Other computer systems suitable for use with the invention may include additional or fewer subsystems. In addition, memory bus 108, peripheral bus 114, and local bus 134 are illustrative of any interconnection scheme serving to link the subsystems. For example, a local bus could be used to connect the CPU to fixed mass storage 116 and display adapter 120. The computer system shown in FIG. 1 is but an example of a computer system suitable for use with the invention. Other computer architectures having different configurations of subsystems may also be utilized.

[0032] As the term is used herein, a tag extension mechanism is a specialized sub-language that enables the addition of new or custom actions, thus allowing the JSP "language" to be easily extended in a portable fashion. A typical example would be elements to support embedded database queries. Tag libraries can be used by JSP authors or JSP authoring tools and can be distributed along with JSP pages to any JSP container, such as Web and application servers. The tag extension mechanism of the present invention can be used from JSP pages written using any valid scripting language, although the mechanism itself only assumes a JAVA™ RunTime environment.

[0033] In general, custom actions provide access to their attribute values and to their body. They can also be nested and their bodies can include scripting elements. For example:

<foo attr1= "val1" attr2= "val2" ...>

[Body]

</foo>

where "Body" may itself have scripting elements or other actions that create or manipulate server side objects.

[0034] In order to provide an extensible multi-lingual tag mechanism, the present invention implements a mechanism similar to a closure, but using three separate and distinct methods.

Under the closure approach, an object on the server side is created that corresponds to a closure, and then the closure is passed to the semantics of the tag, which then invokes a closure. In the present invention, a tag handler, which is the run-time representation of a custom action, has three new methods:

doStart ( )

doBody ( )

doEnd ( )

In other words, the "doIt ( )" method of the closure approach is split into three parts, which can be implemented in JAVA™ (unlike a true closure). The key idea is that an implementation of an action provides a method for start tag, a method for end tag, and a method for body. The three methods interact in a simple way with the page implementation so that the details of the page implementation, including details on how to implement the scripting language, reside in the page.

[0035] In particular, the doStart ( ) method is first invoked with attribute value data. It returns a Boolean value indicating if the body needs to be evaluated. The body is evaluated as a stream of bytes, into an abstraction, known as a BodyEvaluation object. The BodyEvaluation object is then passed into the doBody ( ) method. The doBody ( ) method returns true if more body evaluations are needed. When no more body evaluations are needed, the doEnd ( ) method is evaluated last.

[0036] Thus, rather than creating the complex "closure" abstraction that corresponds to the body, the present invention "in-lines" the evaluation of the body. As shown in Figure 5, the doStart ( ) method processes the start tag and determines if the body needs to be evaluated. If so, a body evaluation buffer is passed to the doBody ( ) method for body evaluation. The doBody ( ) method may be evaluated many times using a different buffer. Once the body evaluation is complete, the doEnd ( ) method completes the processing.

[0037] If a closure were used, one would have to pass the capability of evaluating the body, i.e. create an abstraction that is capable of evaluating body as many times as needed. The code must generally be taken out of order (context). This abstraction is generally very difficult to create, and not really possible using JAVA™. The present invention, however, does not require the creation of the closure abstraction. As shown in Figure 6, the code of Figure 4 can be translated using the present invention into JAVA™ code, without using a closure.

[0038] For the present application, the full power and flexibility of a closure is not required. Note that unlike a closure, the mechanism of the present invention does not require any external context information. The scripting details of the present invention match any nesting of the action, so that the structure corresponding to the original scripting is preserved and the present invention works regardless of the scripting language used. In the present invention, only a body evaluation buffer is needed, and buffers are universally supported.

[0039] In one embodiment, the JSP page implementation instantiates (or reuses) a tag handler object for each action in the JSP page. This handler object is a JAVA™ object that implements the javax.servlet.jsp.tagext.Tag interface. In many cases, the action element has an empty body (no text between the start and the end tag). In this common case, the tag handler's method doStart ( ) is invoked, passing in the values for the attributes to the action and a pageContext object. The pageContext object encapsulates implementation dependent features and provides convenience methods, such as getter methods to obtain references to various request-time objects. The doStart ( ) method then performs the desired action. The pageContext object provides access to all the implicit objects (including the request object, for instance) to the handler.

[0040] When a custom tag introduces some new identifiers, a javax.servlet.jsp.tagext.TagExtraInfo object is involved at JSP translation time (not at

request-time). This object indicates the names and types of the scripting variables that will be assigned objects (at request time) by the action. The only responsibility of the tag author is to indicate this information in the TagExtraInfo object. The corresponding Tag object must add the objects to the pageContext object. It is the responsibility of the JSP translator to automatically supply all the required code to do the "synchronization" between the pageObject values and the scripting variables.

[0041] In general, the body of an action may contain other custom and core tags and scripting elements, as well as uninterpreted template text. If an action element has a non-empty body, two additional methods need to be implemented by the tag handler object. Recall that the doStart ( ) method is always invoked first. This method actually returns a Boolean value that indicates whether the body of the action should be evaluated or not. If so, the body is evaluated into a nested stream that is abstracted as a BodyEvaluation object, then the doBody ( ) method of the tag handler object is invoked with the BodyEvaluation object.

[0042] The doBody ( ) method also returns a Boolean value. In this case, the value indicates whether the JSP page should do further reevaluations of the body text. Note that since server-side objects (accessible via pageContext, or through nested handlers) may have changed, each evaluation may produce very different content for the BodyEvaluation object. When no additional body evaluations are needed, a final doEnd ( ) method is invoked. Like the doStart ( ) method, the doEnd ( ) method provides synchronization actions between the pageContext data and the scripting variables in the JSP page.

[0043] The following examples are provided to further explain the operation of the present invention.

Call Functionality, no Body

<foo att1= "..." att2= " ..." att3= "..." />

In this case, a FooTag tag handler would be defined that implements the Tag interface with null methods for doBody() and doEnd ( ). The doStart ( ) method would take the attribute value information, interact with the pageContext data, and invoke into the appropriate functionality. In this example, no TagExtraInfo information is needed as the action defines no objects accessible to the scripting variables.

Call Functionality, No Body, Define Object

<foo id="myfoo" att1= "..." att2= "..." />

After this, an object with name "myfoo" is available to the scripting language. The type of the object is provided by the TagExtraInfo for "foo." This example is similar to the previous one, except that the doStart ( ) method inserts the appropriate object for the "myfoo" entry.

Iteration

```
<ul>
<foreach iterate= "row" in="balances">
<li>The balance for
account <%= row.getAccount () %> is <%= row.getBalance () %>
</foreach>
</ul>
```

The foreach element does not define a new object for later use but it defines (and redefines) a "row" object that is accessible within its start and end tags. The implementation of this tag requires the repeated evaluation of the body of the tag.

Tag.doStartTag()	This method extracts the value of the "in" and iterates the attribute values. The value of in ("balances" in this example) is used to get at the result data. The value of iterate ("row" in this example) is used as the key on which to store the iteration value. The current value of the "out" variable is stored away so it can be used in doBody(). This method returns true so as to force the evaluation of the body. The translation-time information (TagExtraInfo) indicates that this start tag introduces a scripting variable of name "row".
Tag.doBody()	The BodyEvaluation that is passed in represents the evaluation of the body of the element where the evaluation has been done in a context where the variable "row" is assigned the different rows of the query. This method takes that EvaluationBody and inserts it into the "out" stream that we had saved aside previously.
Tag.doEndTag()	The only thing that needs to be done is some bookkeeping.
TagExtraInfo	Provides translation-time information. In this case, a new scripting variable is declared within the tag but no variables are available after the end tag.

In further detail, a typical action is of the form:

```
<bar id="myBar" foo="one">
    body
</bar>
```

The semantics is carried through a JAVA™ object. This object implements the Tag interface, which has three methods:

doStartTag ( ) The instance is passed attribute values. The Tag instance can use this data to perform any action it may want.

At the end of this action some scripting variables may be assigned from the pageContext object, as indicated in the TagExtraInfo data. This method returns an indication of whether the body of the action should be evaluated; if so, doBody ( ) will be invoked, otherwise doEndTag( ) will be invoked.

doBody ( ) This method is passed a Stream abstraction (i.e. Body evaluation buffer) that corresponds to the tag body. The Tag instance can use this abstraction to insert the data into another Stream, or it can convert it into a String. This method returns an indication of whether the body should be reevaluated. Depending on TagExtraInfo values, the JSP container will resynchronize some variable values (so a reevaluation of the body will return a different value).

doEndTag ( ) This method is invoked after no more doBody() method evaluation arc requested. This gives the Tag handler a last chance to perform cleanup and also the opportunity to update the pageContext object. After this method, the JSP container will update scripting variables as indicated by the TagExtraInfo data.

[0044] In an alternate embodiment, the doStart ( ) method may return an integer instead of a Boolean. For example a "0" may be used to skip the body evaluation, a "1" may be used to require a body evaluation, etc. The doBody ( ) method may also be broken into two separate methods: doInitBody ( ) and doAfterBody ( ). The doInitBody ( ) pre-processes the BodyEvaluation buffer before the first evaluation (i.e. only happens once). Then the

doAfterBody ( ) method functions much like the standard doBody ( ) method discussed above. The doEnd ( ) method can likewise return a value to either skip the processing of the reset of the page (i.e. if an error occurred) or to continue processing the page. Also, instead of passing in attribute values, a JAVA BEAN™ mechanism may be used to set the variable values.

[0045] Those skilled in the art will appreciate that various adaptations and modifications of the just-described preferred embodiment can be configured without departing from the scope and spirit of the invention. Therefore, it is to be understood that, within the scope of the appended claims, the invention may be practiced other than as specifically described herein.